Tabu Search and Genetic Algorithm for Scheduling with Total Flow Time Minimization

Miguel A. González and **Camino R. Vela** and **María Sierra** and **Ramiro Varela**
Dept. of Computer Science and A.I. Centre,
University of Oviedo, 33271 Gijón (Spain)
e-mail: raist@telecable.es, {crvela,sierramaria,ramiro}@uniovi.es

Abstract

In this paper we confront the job shop scheduling problem with total flow time minimization. We start extending the disjunctive graph model used for makespan minimization to represent the version of the problem with total flow time minimization. Using this representation, we adapt local search neighborhood structures originally defined for makespan minimization. The proposed neighborhood structures are used in a genetic algorithm hybridized with a simple tabu search method, outperforming state-of-the-art methods in solving problem instances from several datasets.

Introduction

In this paper we confront the Job Shop Scheduling Problem (JSP) with total flow time minimization. JSP has interested to researches over the last decades, but in most of the cases the objective function is makespan. In (Brucker 2004), two classes of objective functions are considered, termed sum and $bottleneck$ respectively. Objectives of type sum are computed by adding non-decreasing functions of the completion time of the operations, while $bottleneck$ objectives are obtained from the maximum of any of these functions. Total flow time and weigthed tardiness are examples of the first class, and makespan or maximum lateness are examples of the second. In general, problems with sum objectives are harder to solve than their $bottleneck$ counterparts. We observed this fact clearly in (Sierra & Varela 2007), (Sierra & Varela 2008b) and (Sierra & Varela 2008a) through experimental studies across a number of JSP instances, considering both makespan minimization and total flow time minimization. At the same time, objective functions such as total flow time are in many real-life problems more important than it is the makespan. However, researches has paid much more attention to makespan.

We propose a hybrid algorithm that combines a genetic algorithm (GA) with tabu search (TS). The core of this algorithm is a new neighborhood structure that extends some of the neighborhood structures introduced in (Vela, Varela, & González 2009; González, Vela, & Varela 2008) to SDST-JSP (JSP with Sequence Dependent Setup Times), which in its turn extends the structures proposed in (Van Laarhoven, Aarts, & Lenstra 1992) for the classical JSP with makespan minimization. In order to do that, a new disjunctive graph representation for the JSP with total flow time minimization is defined. This representation allows us to establish new results and methods to cope with total flow time minimization. In particular, we have defined a new structure denoted N_F^S. The proposed algorithm is termed $GA + TS - N_F^S$ in the following. We also define a method for estimating the total flow time of the neighbors, and we will see that this estimation is less accurate and more time consuming than similar estimations for the makespan due to the difference in the problem difficulty.

We have conducted an experimental study across conventional benchmarks to compare $GA + TS - N_F^S$ with other state-of-the-art algorithms. In particular, we have considered the heuristic search algorithms proposed in (Sierra & Varela 2008b; Sierra 2009; Sierra & Varela 2010) and the iterative improvement algorithm proposed in (Kreipl 2000). The results shown that the proposed algorithm is quite competitive with both of these methods.

The rest of the paper is organized as follows. In Section (2) we formulate the JSP and introduce the notation used across the paper. In section (3) we summarize the main characteristics of the approaches chosen to compare with. In section (4) we describe the main components of the genetic algorithm. In Section (5), we describe the proposed neighborhood structure, the total flow time estimation algorithm and the main components of the TS algorithm. Section (6) reports results from the experimental study. Finally, in Section (7) we summarize the main conclusions and propose some ideas for future work.

Description of the problem

The JSP requires scheduling a set of N jobs $\{J_1, \ldots, J_N\}$ on a set R of M physical resources or machines $\{R_1, \ldots, R_M\}$. Each job J_i consists of a set of tasks or operations $\{\theta_{i1}, \ldots, \theta_{iM}\}$ to be sequentially scheduled. Each task θ_{ij} has a single resource requirement, a fixed duration $p_{\theta_{ij}}$ and the value of its starting time $st_{\theta_{ij}}$ needs to be determined.

The JSP has two binary constraints: precedence constraints and capacity constraints. Precedence constraints, defined by the sequential routings of the tasks within a job, translate into linear inequalities of the type: $st_{\theta_{ij}} + p_{\theta_{ij}} \leq$

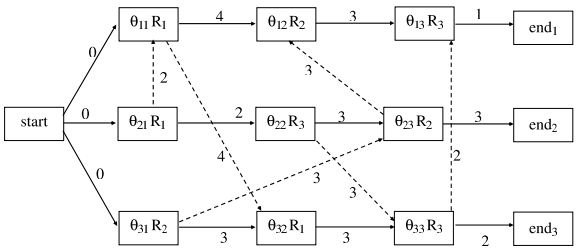

Figure 1: A feasible schedule to a problem with 3 jobs and 3 machines. The total flow time of the schedule is 31.

$st_{\theta_{i(j+1)}}$ (i.e. θ_{ij} before $\theta_{i(j+1)}$). Capacity constraints that restrict the use of each resource to only one task at a time translate into disjunctive constraints of the form: $st_{\theta_{ij}} + p_{\theta_{ij}} \leq st_{\theta_{kl}} \vee st_{\theta_{kl}} + p_{\theta_{kl}} \leq st_{\theta_{ij}}$, where θ_{ij} and θ_{kl} are operations requiring the same machine. The objective is to obtain a feasible schedule such that the total flow time, defined as follows

$$\sum_{i=1,\ldots,N} C_i$$

is minimized, where C_i is the completion time of job i. This problem is denoted by $J||\sum C_i$ according to the $\alpha|\beta|\gamma$ notation used in the literature.

The disjunctive graph model representation

The disjunctive graph is a common representation model for scheduling problems (Roy & Sussmann 1964). The definition of such graph depends on the particular problem. For the $J||\sum C_i$ problem, we propose that it can be represented by a directed graph $G = (V, A \cup E \cup I)$. Each node in the set V represents a task of the problem, with the exception of the dummy nodes *start* and end_i $1 \leq i \leq N$, which represent fictitious operations that do not require any machine. The arcs of A are called *conjunctive arcs* and represent precedence constraints and the arcs of E are called *disjunctive arcs* and represent capacity constraints. The set E is partitioned into subsets E_i, with $E = \cup_{j=1,\ldots,M} E_j$, where E_j corresponds to resource R_j and includes an arc (v, w) for each pair of operations requiring that resource. Each arc (v, w) of A is weighted with the processing time of the operation at the source node, p_v, and each arc (v, w) of E is weighted with p_v. The set I includes arcs (θ_{iM}, end_i), $1 \leq i \leq N$, weighted with $p_{\theta_{iM}}$.

A feasible schedule is represented by an acyclic subgraph G_s of G, $G_s = (V, A \cup H \cup I)$, where $H = \cup_{j=1\ldots M} H_j$, H_j being a hamiltonian selection of E_j. Therefore, finding a solution can be reduced to discovering compatible Hamiltonian selections, i.e. processing orderings for the operations requiring the same resource, or partial schedules, that translate into a solution graph G_s without cycles.

Figure 1 shows a solution to a problem with 3 jobs and 3 machines. Dotted arcs represent the elements of H, while arcs of A are represented by continuous arrows.

The completion time of the job J_i, denoted by C_i, is the cost of the directed path in G_s from node *start* to node end_i having the largest cost. The total flow time of the schedule is then $\sum_{i=1,\ldots,N} C_i$. A *critical path* is a directed path in G_s from node *start* to a node end_i having the largest cost. Nodes and arcs in a critical path are termed *critical*. Each critical path may be represented as a sequence of the form $start, B_1, \ldots, B_r, end_i$ where $1 \leq i \leq N$. Each B_k, $1 \leq k \leq r$, is a maximal subsequence of consecutive operations in the critical path requiring the same machine, called *critical block*. The concepts of critical path and critical block are of major importance for scheduling problems due to the fact that most of the formal properties and solution methods rely on them. For example, most neighborhood structures used in local search algorithms, such as those described in section , consist in reversing arcs in a critical path.

The head of an operation v, denoted r_v, is the cost of the longest path from node *start* to node *v* and it is the starting time of v in the schedule represented by G_s. The tail q_v^i, $1 \leq i \leq N$, is the cost of the longest path from node *v* to node end_i, minus the duration of the task in node *v*. It is easy to see that a node v is critical if and only if $r_v + p_v + q_v^j = C_j$ for some job j. For practical reasons we will take $q_v^i = -\infty$ when no path exist from v to end_i. Here, it is important to remark that we have had to define N tails for each operation, while for makespan minimization it is required just one.

Let PJ_v and SJ_v denote the predecessor and successor of v respectively in the job sequence, and PM_v and SM_v the predecessor and successor of v in its machine sequence. Then, heads and tails are computed as follows. For practical reasons, we consider the node *start* to be PJ_v when v is the first task of its job and PM_v if v is the first task to be executed in a machine. The head of every operation v and every dummy node in the graph may be computed as follows:

$$r_{start} = 0$$
$$r_v = \max(r_{PJ_v} + p_{PJ_v}, r_{PM_v} + p_{PM_v})$$
$$r_{end_i} = r_v + p_v,\ (v, end_i) \in I, 1 \leq i \leq N$$

Also, we consider the node end_i, $1 \leq i \leq N$, as SJ_v if v is the last task of the job i. Then, the tails are computed as follows:

$$q_{end_i}^i = 0$$
$$q_{end_i}^j = -\infty, j \neq i$$
$$q_v^j = \max(q_{SJ_v}^j + p_{SJ_v}, q_{SM_v}^j + p_{SM_v})$$
$$q_{start}^j = \max_{v \in SJ_{start}} \{q_v^j + p_v\}$$

Naturally, the heads have to be computed forward from the *start* node, while the tails have to be computed backwards from the end_i nodes.

Some algorithms for the JSP with total flow time

In this section, we review two previous approaches to the problem: the large step random walk iterative heuristic pro-

posed in (Kreipl 2000) and the best-first heuristic methods proposed in (Sierra 2009).

Large step random walk

In (Kreipl 2000) a local search method based in the large step random walk algorithm is proposed. This method is applied to JSP with weighted tardiness minimization, but this objective function is reduced to total flow time just considering all weights 1 and all due dates 0. The procedure swaps between diversification phases (large steps) and intensification phases (small steps). In large steps the search is guided towards new promising regions of the search space, and these new regions are explored in detail in the small steps. Large steps use a Metropolis algorithm so they can accept worse solutions and escape from a local optimum, while small steps use a hill climbing algorithm so they always reach a local optimum. They use a neighborhood structure previously developed in (Suh 1988), and it's similar to the structures defined in (Taillard 1993) and (Dell' Amico & Trubian 1993) for makespan minimization, as they are based in the concept of critical path. In (Kreipl 2000), the results are compared with the shifting bottleneck heuristic proposed in (Singer & Pinedo 1999), and they obtain better results overall. In our experimental study we have used the implementation of this procedure included in the LEKIN®tool, which is available in *http://www.stern.nyu.edu/om/software/lekin/index.htm*

Heuristic search

We consider the A^* implementation proposed in (Sierra 2009; Sierra, Mencía, & Varela 2009). This is an exact best-first search algorithm that uses an admisible heuristic estimation obtained from relaxations to preemptive one machine sequencing (OMS) problems. One relaxation is made for each machine m and then the heuristic is calculated as:

$$h(n) = \max_{m \in R} \triangle_m; \tag{1}$$

where $\triangle_m$ denotes the optimal cost of the preemptive OMS instance associated to machine m. This value is obtained in polinomial time by means of the algorithm proposed in (Carlier & Pinson 1989; 1994). The A^* algorithm is combined with a powerful method for pruning nodes based on dominance relations among states of the search tree. The resulting algorithm, termed here $A^* - PD$, is able to solve optimally instances up to 10×5 and 9×9. For larger instances, the memory gets usually exhausted before reaching a solution. So, to cope with these situations, in (Sierra 2009), a suboptimal strategy based in heuristic weighting is proposed. This strategy is problem dependent and consist in weighting all the terms $\triangle_m$ of the expression (1) above instead of taking just the largest one. Let us consider that these values are sorted as $\triangle_1 \geq \cdots \geq \triangle_M$, the weighted heuristic function is then computed as:

$$h_{wi}(n) = \triangle_1 + \sum_{2 \leq i \leq M} \frac{\triangle_i}{2^{w_i + \delta}}, w_i > 0. \tag{2}$$

where w_i and δ are parameters. We call this method disjunctive weighting and the resulting algorithm is termed in the sequel as $A^* - DW$. Clearly, $h_{wi}(n) \geq h(n)$. It seems reasonable choosing the values of parameters w_i so as $\triangle_2 \ldots \triangle_M$ to contribute less than $\triangle_1$ to the weighted estimation. In the experiments we have established the following setting: the algorithm starts with parameters $w_i = (i-1)$, $2 \leq i \leq M$ and $\delta = 0$. Then, it iterates over δ at intervals of $0, 2$ until either $\delta = 2$ or the memory gets exhausted. In each iteration, the algorithm finishes when the first solution is reached (which in general is not optimal). Finally, it calculates as many solutions as possible with the largest value of δ that solved the problem without the memory getting exhausted.

Genetic Algorithm for the JSP

The GA used here is taken from (González, Vela, & Varela 2008) and is quite similar to the canonical GA described in the literature; see for example (Holland 1975), (Goldberg 1985) or (Michalewicz 1996). In the first step, the initial population is generated and evaluated. Then the genetic algorithm iterates over a number of steps or generations. In each iteration, a new generation is built from the previous one by applying the genetic operators of selection, recombination and acceptation. These operators can be implemented in a variety of ways and, in principle, are independent from each other. However, in practice all of them should be chosen considering their effect on the remaining ones in order to get a successful overall algorithm. The approach taken in this work is the following. In the selection phase all chromosomes are grouped into pairs, and then each one of these pairs is mated to obtain two offspring. Finally, the acceptation is carried out as a tournament selection among each pair of parents and their two offspring.

The codification schema is based on permutations with repetition as it was proposed by (Bierwirth 1995). In this schema a chromosome is a permutation of the set of operations, each one being represented by its job number. In this way a job number appears within a chromosome as many times as the number of its operations. For example, the chromosome (2 1 1 3 2 3 1 2 3) actually represents the permutation of operations $(\theta_{21}\, \theta_{11}\, \theta_{12}\, \theta_{31}\, \theta_{22}\, \theta_{32}\ \theta_{13}\, \theta_{23}\, \theta_{33})$ and is a valid chromosome for any problem with 3 jobs and 3 machines. This permutation should be understood as expressing partial schedules for each set of operations requiring the same machine. This codification presents a number of interesting characteristics; for example, it is easy to evaluate with different algorithms and allows efficient genetic operators. In (Varela, Serrano, & Sierra 2005) this codification is compared with other permutation based codifications and demonstrated to be the best one for the JSP over a set of 12 selected problem instances of common use.

For chromosome mating we have considered the *Job Order Crossover* (JOX) described in (Bierwirth 1995). Given two parents, JOX selects a random subset of jobs and copies their genes to the offspring in the same positions as they are in the first parent, then the remaining genes are taken from the second parent so as to maintain their relative ordering. We clarify how JOX works by means of an example. Let us consider the following two parents

Parent1 (**2** 1 1 3 **2** 3 1 **2** 3) Parent2 (3 3 1 2 1 3 2 2 1)

If the selected subset of jobs is the one marked in bold (job 2) in the first parent, the generated offspring is

Offspring (**2** 3 3 1 **2** 1 3 **2** 1).

Hence, operator JOX maintains for each machine a subsequence of operations in the same order as they are in parent 1 while the remaining operations keep the same order as in parent 2, but their positions in general change. The operator JOX might swap any two operations requiring the same machine; this is an implicit mutation effect. For this reason, we have not used any explicit mutation operator. So, parameter setting in experimental study is considerably simplified, as crossover probability is set to 1 and mutation probability need not be specified. Of course, for identical parent sequences, the offspring will be identical and consequently the evolution would come to a complete halt if all chromosomes were identical. However, in practice this is not an issue as the algorithm always stops before convergence to such situation. With this setting, we have obtained results quite similar to those obtained with a lower crossover probability and a low probability of applying conventional order based mutation operators.

To build schedules we have used a decoding algorithm that generates active schedules. A schedule is active if no operation can be started earlier without delaying any other operation. In the implementation we used the Serial Schedule Generation Schema (SSGS) proposed in (Artigues, Lopez, & Ayache 2005) for the JSP with setup times. SSGS iterates over the operations in the chromosome sequence and assigns each the earliest starting time that satisfies all constraints with respect to the previous scheduled operations.

When combined with GA, TS is applied to every schedule produced by SSGS. Then, the chromosome is rebuilt from the improved schedule obtained by TS, so as its characteristics can be transferred to subsequent offsprings. This effect of the evaluation function is known as Lamarckian evolution.

Tabu Search for the Total Flow Time minimization in the JSP

Algorithm 1 shows the tabu search algorithm considered herein. This algorithm is borrowed from (González, Vela, & Varela 2009), and it similar to other tabu search algorithms described in the literature (Glover & Laguna 1997). In the first step the initial solution (i.e. a chromosome generated by the GA, after applying an active schedule builder) is evaluated. Then, it iterates over a number of steps. In each iteration, the neighborhood of the current solution is built and one of the neighbors is selected for the next iteration. The tabu search stops after performing a given number of iterations *maxGlobalIter*, returning the best solution reached so far. In order to avoid reevaluating the same solutions, the algorithm uses tabu tenure and cycle checking mechanisms.

input An initial solution s_0 for a problem instance *P*
output A (hopefully improved) solution s_B for instance *P*
  Set the current solution $s = s_0$ and the best solution $s_B = s$;
  Set *globalIter* = 0, Empty the tabu list;
  while *globalIter* $<$ *maxGlobalIter* **do**
    Set *globalIter* $=$ *globalIter*+1;
    Generate neighbors of the current solution s by means of the neighborhood structure;
    Let $s*$ be the best neighbor either not tabu and not leading to a cycle or satisfying the aspiration criterion. Update the tabu list and the cycle detection structure accordingly and let $s = s*$;
    if $s*$ is better than s_B **then**
      Set $s_B = s*$;
  return The solution s_B;

Alg. 1: The Tabu Search Algorithm

The neighborhood structure

The neighborhood structure N_F^S proposed below is adapted from that termed N^S in (González, Vela, & Varela 2008; 2009). N^S was defined for JSP with setup times and makespan minimization and it is in turn based on previous structures given in (Matsuo, Suh, & Sullivan 1988) and (Van Laarhoven, Aarts, & Lenstra 1992) for the standard JSP. These structures have given rise to some of the most outstanding algorithms for the JSP such as, for example, those proposed in (Dell' Amico & Trubian 1993; Nowicki & Smutnicki 2005; Balas & Vazacopoulos 1998; Zhang *et al.* 2008).

As it is usual, N_F^S is based on reversing arcs in a critical path, so a condition for feasibility is required after a move. In our implementation we have used the following theorem. Its proof is quite similar to that of an analogous theorem given in (Vela, Varela, & González 2009) for the JSP with makespan minimization.

Theorem 1. *Given a critical block of the form* $(b'\ v\ b\ w\ b'')$*, where* b*,* b' *and* b'' *are sequences of operations, a sufficient condition for an alternative path from* v *to* w *not to exist is that*

$$\forall u \in \{v\} \cup b, r_{PJ_w} < r_{SJ_u} + p_{SJ_u} \tag{3}$$

Then, the neighborhood structure N_F^S is defined as follows.

Definition 1 (N_F^S)**.** *Let operation* v *be a member of a critical block* B*. In a neighboring solution* v *is moved to another position in* B*, provided that the sufficient condition of feasibility (3) is preserved.*

In principle, N critical paths should be considered in order to generate neighbors, i.e. one critical path for each node end_i. However, it is possible to consider less critical paths (for example the largest ones), so reducing the number of neighbors. In any case, some mechanism to avoid the repetition of neighbors is necessary as critical paths from different nodes end_i have usually some parts in common.

Table 1: Results of GA+TS across instance LA02 exploiting different numbers of critical paths

Critical Paths	Best	Average	Time(s.)
1	4545(1)	4574.7	32
5 (half)	4480(3)	4490.9	37
10 (all)	4459(2)	4471.2	41

Table 1 shows results from some experiments across LA02 instance (10 jobs and 5 machines), launching 15 runs of the GA+TS algorithm with a configuration of $10 \times 15 \times 1500$ (GA population size $\times$ GA number of generations $\times$ TS maxGlobalIter), exploiting a different number of critical paths in each experiment: the largest one, the 5 largest ones or all of them. The first column shows the number of jobs considered to build the critical paths, the second column shows the best result in all 15 runs (in parentheses it is the number of times that the best solution is reached), the third column shows the average solution obtained and the fourth column shows the average time taken. As we can observe the best choice is to exploit all the N critical paths, so as the largest number of neighbors is evaluated. Only in this case the GA+TS algorithm reaches the optimal solution (4459), while the time taken augments in about 25%. We have conducted experiments across other instances with similar results.

Total flow time estimation

Even though computing the total flow time of a neighbor only requires to recompute heads (tails) of operations that are after (before) the first (last) operation moved, for the sake of efficiency the selection rule is based on total flow time estimations instead of computing the actual total flow time of all neighbors. For this purpose, we have extended the procedure *lpath* given for the JSP in (Taillard 1993). This procedure is termed *lpathTFT* and it is shown in Algorithm 2. It takes an input sequence of operations of the form $(Q_1 \dots Q_q)$ after a move, all of them requiring the same machine, being $(Q_1 \dots Q_q)$ a permutation of the sequence $(O_1 \dots O_q)$ before the move. The algorithm works as follows: For each $i = 1 \dots N$, *lpathTFT* estimates the cost of the longest path from node $start$ to node end_i through a node included in $(Q_1 \dots Q_q)$, this estimation is given by $\max_{j=1\dots q} \{r'_{Q_j} + p_{Q_j} + q'^i_{Q_j}$, where $q'^i_{Q_j}$ is the tail of node Q_j corresponding to node end_i after the move (remember that each operation has a tail for each one of the end_i nodes) and then adds up the estimations from all the paths to obtain the final estimated total flow time of the neighboring schedule. When w is moved before v in a block of the form $(b'\ v\ b\ w\ b'')$, the input sequence is $(w\ v\ b)$, and if v is moved after w the input sequence is $(b\ w\ v)$.

The makespan estimation algorithm *lpath* is very accurate and very efficient. However, estimation for the total flow time is much more time consuming as it calculates N tails for each operation. Moreover, experiments conclude that total flow time estimation is much less accurate than makespan estimation, as Table 2 shows. This table reports the percentage of times that the estimations are greater, equal or lower than the actual values from similar GA+TS algorithms for makespan and total flow time respectively (running with the same parameters as the experiments reported in Table 1). As it can be expected, the number of neighbors evaluated is much larger for total flow time than it is for makespan and the estimations are much more accurate for the makespan. In any case, a remarkable result is that only in a small fraction of the cases the estimation is larger than the actual total flow time.

As the ratio of underestimations is really high (34.75%), and estimation error is also larger than it is in the makespan case, we have opted to evaluate the actual total flow time when the neighbor's estimation is lower than the actual total flow time of the original schedule. Some preliminary results have shown that the improvement achieved in this way makes up the time consumed.

input A sequence of operations $(Q_1 \dots Q_q)$ as they appear after a move
output A estimation of the total flow time of the resulting schedule
$Est = 0$;
$a = Q_1$;
$r'_a = \max\{r_{PJ_a} + p_{PJ_a}, r_{PM_a} + p_{PM_a}\}$;
for $i = 2$ to q **do**
  $b = Q_i$;
  $r'_b = \max\{r_{PJ_b} + p_{PJ_b}, r'_a + p_a\}$;
  $a = b$;
for $i = 1$ to N **do**
  $b = Q_q$;
  $q'^i_b = \max\{q^i_{SJ_b} + p_{SJ_b}, q^i_{SM_b} + p_{SM_b}\}$;
  for $j = q - 1$ to 1 **do**
    $a = Q_j$;
    $q'^i_a = \max\{q^i_{SJ_a} + p_{SJ_a}, q'^i_b + p_b\}$;
    $b = a$;
  $Est = Est + \max_{j=1\dots q}\{r'_{Q_j} + p_{Q_j} + q'^i_{Q_j}\}$;
return Est;

Alg. 2: Procedure *lpathTFT* $(Q_1 \dots Q_q)$

Table 2: Accuracy of the estimation algorithms for makespan and total flow time in LA02 instance

Function	Estimations	$>$	$=$	$<$
Makespan	77 millions	0.36%	95.77%	3.87%
T. F. T.	192 millions	0.73%	64.52%	34.75%

Experimental Study

The purpose of the experimental study is to compare $GA + TS - N_F^S$ with other state-of-the-art algorithms. Firstly, we consider the exact A* algorithm enhanced with a pruning by dominance method (A^*-PD) and a sub-optimal variant of this algorithm that uses a method of heuristic weighting (A^*-DW), both of them proposed in (Sierra 2009). Also, we consider the large step random walk local search algorithm (LSRW) proposed in (Kreipl 2000).

We have conducted two series of experiments on standard benchmarks taken from the OR-library. In the first one, we experimented across a set of instances (sizes 10×5, 8×8 and 9×9) proposed in (Sierra 2009), that are optimally solved by A^*-PD. The reduced 8×8 and 9×9 instances are built from original 10×10 instances removing the last jobs and the last machines. In the second series of experiments we used larger instances (sizes 15×5, 20×5 and 10×10) that can not be optimally solved by A^*-PD. In all these experiments, we have to be aware of the differences in the target machines and then in the time taken. The versions of A^* have been run on Ubuntu V8,04 on Intel Core 2 Duo at 2,13GHz with 7,6Gb of RAM, LSRW has been run on Windows XP on Intel Core 2 Duo at 2,13GHz with 3Gb of RAM and $GA+TS-N_F^S$ has been run on Windows XP on Intel Core 2 Duo at 2.66GHz with 2Gb of RAM.

Table 3 shows results from the first experiments. In this case A^*-PD reach the optimal solutions for all instances. LSRW and $GA+TS-N_F^S$ were run 20 times for each instance and the best and average values of all 20 solutions are reported. The $GA+TS-N_F^S$ parameters (GA population size $\times$ GA number of generations $\times$ TS *maxGlobalIter*) were $(50\times70\times150)$ for both 10×5 and 9×9 instances, and $(30\times40\times120)$ for 8×8 instances. With these values the algorithm converges properly and the time taken is not larger than that of the other two algorithms (taking into account the target machines).

As we can observe, $GA+TS-N_F^S$ is able to reach the optimum solution in most of the trials: it reached the optimal solution in 747 out of the 820 runs for all 41 instances (91.1%), and has reached at least once the optimal solution for 40 of the 41 instances. The exception is the instance ORB08(9×9) where A^*-PD takes the largest time among the 9×9 instances. LSRW fails to reach the optimal solution in 6 of the 41 instances.

In the second series of experiments we compare A^*-DW, LSRW and $GA+TS-N_F^S$ on the set of instances LA06-10 (15×5), LA11-15 (20×5) and LA15-20 (10×10). None of these instances can be optimally solved by A^*-PD before the memory getting exhausted, so we do not know their optimal solutions. In these experiments the $GA+TS-N_F^S$ parameters were $(50\times70\times150)$ for all 15 instances, run.

Table 4 shows the results from these experiments. Here, the time reported for A^*-DW corresponds to the number of trials that are required to adjust the parameter δ to its best value and one more to obtain all possible solutions with this parameter, the time taken until the memory getting exhausted or the whole search space is explored is accumulated for all trials. For LWRS and $GA+TS-N_F^S$ the time is the average time of 20 trials.

As we can observe, in average $GA+TS-N_F^S$ is better than LSRW in 12 instances and it is worse in 3; and the best value of $GA+TS-N_F^S$ is better than that of LSRW in 11 and equal in 4. Compared to A^*-DW, $GA+TS-N_F^S$ is in average better in 7 cases, equal in 5 and worse in 3, and the best value of $GA+TS-N_F^S$ is better in 8 cases and equal in the remaining ones.

Conclusions

We have considered the job shop scheduling problem, where the objective is to minimize the total flow time. We have proposed a disjunctive graph representation for this problem and used it to define a specific neighborhood structure for the total flow time. The neighborhood structure has then been used in a tabu search algorithm, which is embedded in a genetic algorithm framework. We have also defined a method for estimating the total flow time of the neighbors, and demonstrated that estimating this objective function is much more difficult than estimating other classic objective functions such as the makespan.

We have reported results from an experimental study across some conventional benchmarks and compared the proposed $GA+TS-N_F^S$ algorithm with some representative state-of-the-art methods: the A^* algorithm with a pruning by dominance method (A^*-PD) and the A^*-DW, both proposed in (Sierra 2009), and the large step random walk algorithm (LSRW) proposed in (Kreipl 2000). The results show that the proposed approach is competitive with these state-of-the-art methods.

As future work we plan to extend our approach to confront other variants or extensions of this problem. We would like to consider the weighted tardiness as objective function, which is very interesting too in real life applications, and it is a generalization of total flow time. It would also be interesting to tackle multiobjective problems. Finally, our approach may also be applied to more general frameworks than JSP, such as resource-constrained scheduling with setup times.

Acknowledgments

This work is supported by MEC-FEDER Grant TIN2007-67466-C02-01 and FICYT grant BP07-109.

References

Artigues, C.; Lopez, P.; and Ayache, P. 2005. Schedule generation schemes for the job shop problem with sequence-dependent setup times: Dominance properties and computational analysis. *Annals of Operations Research* 138:21–52.

Balas, E., and Vazacopoulos, A. 1998. Guided local search with shifting bottleneck fo job shop scheduling. *Management Science* 44(2):262–275.

Bierwirth, C. 1995. A generalized permutation approach to jobshop scheduling with genetic algorithms. *OR Spectrum* 17:87–92.

Brucker, P. 2004. *Scheduling Algorithms*. Springer, 4th edition.

Carlier, J., and Pinson, E. 1989. An algorithm for solving the job-shop problem. *Management Science* 35(2):164–176.

Carlier, J., and Pinson, E. 1994. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research* 78:146–161.

Dell' Amico, M., and Trubian, M. 1993. Applying tabu search to the job-shop scheduling problem. *Annals of Operational Research* 41:231–252.

Glover, F., and Laguna, M. 1997. *Tabu Search*. Kluwer Academic Publishers.

Goldberg, D. 1985. *Genetic algorithms in search. Optimization and machine learning*. Addison-Wesley.

González, M. A.; Vela, C. R.; and Varela, R. 2008. A new hybrid genetic algorithm for the job shop scheduling problem with setup times. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS-2008)*. Sidney: AAAI Press.

González, M. A.; Vela, C. R.; and Varela, R. 2009. Genetic algorithm combined with tabu search for the job shop scheduling problem with setup times. In *IWINAC 2009: Methods and Models in Artificial and Natural Computation*, 265–274. LNCS-5601, Springer.

Holland, J. H. 1975. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. The University of Michigan Press.

Kreipl, S. 2000. A large step random walk for minimizing total weighted tardiness in a job shop. *Journal of Scheduling* 3:125–138.

Matsuo, H.; Suh, C.; and Sullivan, R. 1988. A controlled search simulated annealing method for the general jobshop scheduling problem. Working paper 03-44-88, Graduate School of Business, University of Texas.

Michalewicz, Z. 1996. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, third, revised and extended edition.

Nowicki, E., and Smutnicki, C. 2005. An advanced tabu search algorithm for the job shop problem. *Journal of Scheduling* 8:145–159.

Roy, B., and Sussmann, B. 1964. Les problèmes d'ordonnancement avec constraintes disjonctives. Note d.s. no. 9 bis, d6c, SEMA, Matrouge, Paris.

Sierra, M., and Varela, R. 2007. Pruning by dominance in best-first search. In *Proceedings of CAEPIA'2007*, volume 2, 289–298.

Sierra, M., and Varela, R. 2008a. A new admissible heuristic for the job shop scheduling problem with total flow time. *ICAPS-2008. Workshop on Constraint Satisfaction Techniques for Planning and Scheduling. Sidney*.

Sierra, M., and Varela, R. 2008b. Pruning by dominance in best-first search for the job shop scheduling problem with total flow time. *Journal of Intelligent Manufacturing, DOI 10.1007/s10845-008-0167-4* 1:1–2.

Sierra, M. R., and Varela, R. 2010. Best-first search and pruning by dominance for the job shop scheduling problem with total flow time. *Journal of Intelligent Manufacturing* 21(1):111–119.

Sierra, M. R.; Mencía, C.; and Varela, R. 2009. Weighting disjunctive heuristics for scheduling problems with summation cost functions. In *Proceedings of Workshop on Planning, Scheduling and Constraint Satisfaction, CAEPIA'2009*.

Sierra, M. R. 2009. *Métodos de Poda por Dominancia en Búsqueda Heurística. Aplicaciones a Problemas de Scheduling*. Ph.D. Dissertation, Universidad de Oviedo, Spain.

Singer, M., and Pinedo, M. 1999. A shifting bottleneck heuristic for minimizing the total weighted tardiness in a job shop. *Naval Research Logistics* 46(1):1–17.

Suh, C. 1988. *Controlled search simulated annealing for job shop scheduling*. Ph.D. Dissertation, University of Texas.

Taillard, E. 1993. Benchmarks for basic scheduling problems. *European Journal of Operational Research* 64:278–285.

Van Laarhoven, P.; Aarts, E.; and Lenstra, K. 1992. Job shop scheduling by simulated annealing. *Operations Research* 40:113–125.

Varela, R.; Serrano, D.; and Sierra, M. 2005. New codification schemas for scheduling with genetic algorithms. *Proceedings of IWINAC 2005. Lecture Notes in Computer Science* 3562:11–20.

Vela, C. R.; Varela, R.; and González, M. A. 2009. Local search and genetic algorithm for the job shop scheduling problem with sequence dependent setup times. *Journal of Heuristics* DOI 10.1007/s10732-008-9094-y.

Zhang, C. Y.; Li, P.; Rao, Y.; and Guan, Z. 2008. A very fast TS/SA algorithm for the job shop scheduling problem. *Computers and Operations Research* 35:282–294.

Table 3: Results from A^*-PD, LSRW and $GA + TS - N_F^S$.

Instance	Size	A^*-PD		LSRW			$GA + TS - N_F^S$		
		Optimum	Time(s)	Best	Avg.	Time(s)	Best	Avg.	Time(s)
LA01	10×5	4832	35	4832(1)	4832.9	93	4832(20)	4832	90
LA02	10×5	4459	80	4479(4)	4483.2	93	4459(20)	4459	96
LA03	10×5	4151	10	4151(20)	4151	93	4151(20)	4151	101
LA04	10×5	4259	19	4259(2)	4268.8	93	4259(20)	4259	96
LA05	10×5	4072	68	4072(2)	4095.0	93	4072(20)	4072	108
LA16	8×8	4600	4	4600(3)	4606.5	17	4600(20)	4600	16
LA17	8×8	4366	6	4379(20)	4379	17	4366(20)	4366	15
LA18	8×8	4690	3	4690(13)	4704.7	17	4690(17)	4696.3	16
LA19	8×8	4612	3	4612(20)	4612	17	4612(19)	4613.8	14
LA20	8×8	4616	5	4616(20)	4616	17	4616(20)	4616	16
ORB01	8×8	4743	4	4743(20)	4743	17	4743(20)	4743	16
ORB02	8×8	4678	5	4678(20)	4678	17	4678(20)	4678	17
ORB03	8×8	4925	10	4925(20)	4925	17	4925(20)	4925	16
ORB04	8×8	5081	4	5081(20)	5081	17	5081(20)	5081	18
ORB05	8×8	4191	3	4191(5)	4192.5	17	4191(20)	4191	17
ORB06	8×8	4673	11	4673(20)	4673	17	4673(20)	4673	16
ORB07	8×8	2124	9	2124(20)	2124	17	2124(19)	2124.1	18
ORB08	8×8	4749	40	4749(6)	4759.9	17	4749(13)	4753.6	17
ORB09	8×8	4590	20	4590(20)	4590	17	4590(20)	4590	20
ORB10	8×8	4959	1	4959(20)	4959	17	4959(20)	4959	14
ABZ5	8×8	6818	3	6839(4)	6891	17	6818(20)	6818	16
ABZ6	8×8	4900	4	4900(2)	4922.5	17	4900(20)	4900	16
FT10	8×8	4559	4	4559(20)	4559	17	4559(20)	4559	16
LA16	9×9	5724	38	5724(6)	5739.6	294	5724(20)	5724	76
LA17	9×9	5390	116	5396(5)	5403.5	294	5390(20)	5390	82
LA18	9×9	5770	34	5770(20)	5770	294	5770(20)	5770	90
LA19	9×9	5891	28	5891(20)	5891	294	5891(20)	5891	68
LA20	9×9	5915	110	5934(12)	5935.2	294	5915(20)	5915	73
ORB01	9×9	6367	166	6367(5)	6378.5	294	6367(8)	6371.6	75
ORB02	9×9	5867	92	5867(3)	5867.9	294	5867(8)	5867.6	78
ORB03	9×9	6310	110	6310(20)	6310	294	6310(20)	6310	81
ORB04	9×9	6661	273	6661(20)	6661	294	6661(3)	6676.3	85
ORB05	9×9	5605	16	5605(20)	5605	294	5605(20)	5605	88
ORB06	9×9	6106	208	6106(20)	6106	294	6106(20)	6106	78
ORB07	9×9	2668	155	2668(20)	2668	294	2668(20)	2668	86
ORB08	9×9	5656	772	5668(2)	5693.3	294	5668(19)	5670.5	84
ORB09	9×9	6013	38	6013(18)	6013.8	294	6013(20)	6013	99
ORB10	9×9	6328	106	6328(1)	6332.75	294	6328(20)	6328	89
ABZ5	9×9	8586	39	8586(20)	8586	294	8586(20)	8586	79
ABZ6	9×9	6524	29	6524(14)	6524.6	294	6524(20)	6524	91
FT10	9×9	5982	72	5982(20)	5982	294	5982(20)	5982	78

Table 4: Results from A^*-DW, LSRW and $GA + TS - N_F^S$.

Instance	Size	A^*-DW		LSRW			$GA + TS - N_F^S$		
		Best	Time(s)	Best	Avg.	Time(s)	Best	Avg.	Time(s)
LA06	15×5	8631	859	8644(1)	8670.9	840	8625(1)	8628	307
LA07	15×5	8069	1005	8116(1)	8165.9	840	8069(19)	8070.7	295
LA08	15×5	8190	732	7949(8)	7960.7	840	7946(10)	7962.4	328
LA09	15×5	9153	583	9113(1)	9186.6	840	9034(3)	9072.6	345
LA10	15×5	8798	763	8821(1)	8881.7	840	8798(12)	8799.6	315
LA11	20×5	14014	657	14148(2)	14196.4	840	13880(1)	13985.5	715
LA12	20×5	12594	501	11733(1)	11819	840	11710(3)	11753.1	895
LA13	20×5	13495	538	13477(1)	13558.1	840	13281(1)	13367.6	774
LA14	20×5	14556	595	14671(1)	14738.7	840	14514(1)	14573.4	743
LA15	20×5	14279	519	14285(1)	14380.0	840	14111(1)	14187.4	819
LA16	10×10	7376	2143	7376(19)	7376.5	840	7376(20)	7376	111
LA17	10×10	6537	2439	6537(1)	6566.8	840	6537(20)	6537	106
LA18	10×10	6970	3829	6970(1)	7005.0	840	6970(20)	6970	108
LA19	10×10	7217	1503	7217(15)	7217.7	840	7217(15)	7223.3	98
LA20	10×10	7345	1351	7394(15)	7397.4	840	7345(7)	7402.4	98